InDepth

Distributed Caching with Memcached

Cut the load on your Web site's database by adding a scalable object caching layer to your application.

by Brad Fitzpatrick

Memcached is a high-performance, distributed caching system. Although application-neutral, it's most commonly used to speed up dynamic Web applications by alleviating database load. Memcached is used on LiveJournal, Slashdot, Wikipedia and other high-traffic sites.

Motivation

For the past eight years I've been creating large, interactive, database-backed Web sites spanning multiple servers. Approximately 70 machines currently run LiveJournal.com, a blogging and social networking system with 2.5 million accounts. In addition to the typical blogging and friend/interest/profile declaration features, LiveJournal also sports forums, polls, a per-user news aggregator, audio posts by phone and other features useful for bringing people together.

Optimizing the speed of dynamic Web sites is always a challenge, and LiveJournal is no different. The task is made all the more challenging, because nearly any content item in the system can have an associated security level and be aggregated into many different views. From prior projects with dynamic, context-aware content, I knew from the beginning of LiveJournal's development that pregenerating static pages wasn't a viable optimization technique. It's impossible due to the constituent objects' cacheability and lifetimes being so different, so you make a bunch of sacrifices and waste a lot of time precomputing pages more often than they're requested.

This isn't to say caching is a bad thing. On the contrary, one of the core factors of a computer's performance is the speed, size and depth of its memory hierarchy. Caching definitely is necessary, but only if you do it on the right medium and at the right granularity. I find it best to cache each object on a page separately, rather than caching the entire page as a whole. That way you don't end up wasting space by redundantly caching objects and template elements that appear on more than one page.

In the end, though, it's all a series of trade-offs. Because processors keep getting faster, I find it preferable to burn CPU cycles rather than wait for disks. Modern disks keeping growing larger and cheaper, but they aren't getting much faster. Considering how slow and crash-prone they are, I try to avoid disks as much as possible. LiveJournal's Web nodes are all diskless, Netbooting off a common yet redundant NFS root image. Not only is this cheaper, but it requires significantly less maintenance.

Of course, disks are necessary for our database servers, but there we stick to fast disks with fast RAID setups. We actually have ten different database clusters, each with two or more machines. Nine of the clusters are user clusters, containing data specific to the users partitioned among them. One is our global cluster with non-user data and the table that maps users to their user clusters. The rationale for independent clusters is to spread writes. The alternative is having one big cluster with hundreds of slaves. The difficulty with such a monolithic cluster is it only spreads reads. The problem of diminishing returns appears as each new slave is added and increasingly is consumed by the writes necessary to stay up to date.
At this point you can see LiveJournal's back-end philosophy:

1. Avoid disks: they're a pain. When necessary, use only fast, redundant I/O systems.

2. Scale out, not up: many little machines, not big machines.

My definition of a little machine is more about re-usability than cost. I want a machine I can keep using as long as it's worth its space and heat output. I don't want to scale by throwing out machines every six months, replacing them with bigger machines.

**Where to Cache?**

Prior to Memcached, our Web nodes unconditionally hit our databases. This worked, but it wasn't as optimal as it could've been. I realized that even with 4G or 8G of memory, our database server caches were limited, both in raw memory size and by the address space available to our database server processes running on 32-bit machines. Yes, I could've replaced all our databases with 64-bit machines with much more memory, but recall that I'm stubborn and frugal.

I wanted to cache more on our Web nodes. Unfortunately, because we're using mod_perl 1.x, caching is a pain. Each process and thus, each Web request, is in its own address space and can't share data with the others. Each of the 30–50 processes could cache on its own, but doing so would be wasteful.

System V shared memory has too many weird limitations and isn't portable. It also works only on a single machine, not across 40+ Web nodes. These issues reflect what I saw as the main problem with most caching solutions. Even if our application platform was multithreaded with data easily shared between processes, we still could cache on only a single machine. I didn't want all 40+ machines caching independently and duplicating information.

**Memcached Is Born**

One day, sick of how painful it is to cache efficiently in mod_perl applications, I started dreaming. I realized we had a lot of spare memory available around the network, and I wanted to use it somehow. If you're a Perl programmer strolling through CPAN, you find an abundance of Cache::* modules. The interface to almost all of them is a dictionary. If you're fortunate enough to have missed Computer Science 101, a dictionary is the name of the abstract data type that maps keys to values. Perl people call that an associative array or a hash, short for hash table. A hash table is a specific type of data structure that provides a dictionary interface.

I wanted a global hash table that all Web processes on all machines could access simultaneously, instantly seeing one another's changes. I'd use that for my cache. And because memory is cheap, networks are fast and I don't trust servers to stay alive, I wanted it spread out over all our machines. I did a quick search, found nothing and started building it.

Each Memcached server instance listens on a user-defined IP and port. The basic idea is you run Memcached instances all over your network, wherever you have free memory and your application uses them all. It's even useful to run multiple instances on the same machine, if that machine is 32-bit and has more total memory than the kernel makes available to a single process. For example, while we were learning our lesson on scaling out and not up, we picked up a ridiculously expensive machine that happens to have 12GB of memory. Nowadays, we use it for a number of miscellaneous tasks, one of which is running five 2GB Memcached instances. That gives us 10GB more memory in our global cache from a single machine, even though each process on 32-bit Linux usually can address only 3GB of memory.
The trick to Memcached is that for a given key, it needs to pick the same Memcached node consistently to handle that key, all while spreading out storage (keys) evenly across all nodes. It wouldn't work to store the key foo on machine 1 and then later have another process try to load foo from machine 2. Fortunately, this isn't a hard problem to solve. We simply can think of all the Memcached nodes on the network as buckets in a hash table.

How Memcached Works

Figure 1. The Memcached client library is responsible for sending requests to the correct servers.

Step 1: the application requests keys foo, bar and baz using the client library, which calculates key hash values, determining which Memcached server should receive requests.

Step 2: the Memcached client sends parallel requests to all relevant Memcached servers.

Step 3: the Memcached servers send responses to the client library.

Step 4: the Memcached client library aggregates responses for the application.

If you know how a hash table works, skim along. If you're new to hashes, here's a quick overview. A hash table is implemented as an array of buckets. Each bucket (array element) contains a list of nodes, with each
node containing \([key, value]\). This list later is searched to find the node containing the right key. Most hashes start small and dynamically resize over time as the lists of the buckets get too long.

A request to get/set a key with a value requires that the key be run through a hash function. A hash function is a one-way function mapping a key (be it numeric or string) to some number that is going to be the bucket number. Once the bucket number has been calculated, the list of nodes for that bucket is searched, looking for the node with the given key. If it's not found, a new one can be added to the list.

So how does this relate to Memcached? Memcached presents to the user a dictionary interface \((key -> value)\), but it's implemented internally as a two-layer hash. The first layer is implemented in the client library; it decides which Memcached server to send the request to by hashing the key onto a list of virtual buckets, each one representing a Memcached server. Once there, the selected Memcached server uses a typical hash table.

Each Memcached instance is totally independent, and does not communicate with the others. Each instance drops items used least recently by default to make room for new items. The server provides many statistics you can use to find query/hit/miss rates for your entire Memcached farm. If a server fails, the clients can be configured to route around the dead machine or machines and use the remaining active servers. This behavior is optional, because the application must be prepared to deal with receiving possibly stale information from a flapping node. When off, requests for keys on a dead server simply result in a cache miss to the application. With a sufficiently large Memcached farm on enough unique hosts, a dead machine shouldn't have much impact on global hit rates.

**Our Setup**

LiveJournal.com currently has 28 Memcached instances running on our network on ten unique hosts, caching the most popular 30GB of data. Our hit rate is around 92%, which means we're hitting our databases a lot less often than before.

On our Web nodes with 4GB of memory, we run three Memcached instances of 1GB each, then mod_perl using 500MB, leaving 500MB of breathing room. Running Memcached on the same machine as mod_perl works well, because our mod_perl code is CPU-heavy, whereas Memcached hardly touches the CPU. Certainly, we could buy machines dedicated to Memcached, but we find it more economical to throw up Memcached instances wherever we happen to have extra memory and buy extra memory for any old machine that can take it.

You even can run a Memcached farm with all instances being different sizes. We run a mix of 512MB, 1GB and 2GB instances. You can specify the instances and their sizes in the client configuration, and the Memcached connection object weights appropriately.

**Speed**

Of course, the primary motivation for caching is speed, so Memcached is designed to be as fast as possible. The initial prototype of Memcached was written in Perl. Although I love Perl, the prototype was laughably slow and bloated. Perl trades off memory usage for everything, so a lot of precious memory was wasted, and Perl can't handle tons of network connections at once.

The current version is written in C as a single-process, single-threaded, asynchronous I/O, event-based daemon. For portability and speed, we use libevent (see the on-line Resources section) for event notification. The advantage of libevent is that it picks the best available strategy for dealing with file descriptors at runtime. For example, it chooses kqueue on BSD and epoll on Linux 2.6, which are efficient when dealing with thousands of concurrent connections. On other systems, libevent falls back to the traditional poll and
select methods.

Inside Memcached, all algorithms are O(1). That is, the runtime of the algorithms and CPU used never varies with the number of concurrent clients, at least when using kqueue or epoll, or with the size of the data or any other factor.

Of note, Memcached uses a slab allocator for memory allocation. Early versions of Memcached used the malloc from glibc and ended up falling on their faces after about a week, eating up a lot of CPU space due to address space fragmentation. A slab allocator allocates only large chunks of memory, slicing them up into little chunks for particular classes of items, then maintaining freelists for each class whenever an object is freed. See the Bonwick paper in Resources for more details. Memcached currently generates slab classes for all power-of-two sizes from 64 bytes to 1MB, and it allocates an object of the smallest size that can hold a submitted item. As a result of using a slab allocator, we can guarantee performance over any length of time. Indeed, we've had production Memcached servers up for 4–5 months at a time, averaging 7,000 queries/second, without problems and maintaining consistently low CPU usage.

Another key requirement for Memcached was that it be lockless. All objects are multiversioned internally and reference counted, so no client can block any other client's actions. If one client is updating an object stored in Memcached while a dozen others are downloading it, even with one client on a lossy network connection dropping half its packets, nobody has to wait for anybody else.

A final optimization worth noting is that the protocol allows fetching multiple keys at once. This is useful if your application knows it needs to load a few hundred keys. Instead of retrieving them all sequentially, which would take a fraction of a second in network round-trips, the application can fetch them all in one request. When necessary, the client libraries automatically split multi-key loads from the application into separate parallel multi-key loads to the Memcached instances. Alternatively, applications can provide explicit hash values with keys to keep groups of data on the same instance. That also saves the client library a bit of CPU time by not needing to calculate hash values.

**Client Libraries**

The client/server interface to Memcached is simple and lightweight. As such, there are client libraries for Perl, PHP, Python and Java. I also hear that a coworker of mine has been working on a Ruby client, due out soon.

All of the clients support object serialization using their native serialization methods. Perl uses Storable, PHP uses serialize, Python uses Pickle and Java uses the Serializable interface. Most clients also support transparent compression, optionally only past a certain size threshold. Both serialization and compression are possible because Memcached lets client modules store opaque flags alongside stored items, indicating how they should handle the data coming out.

**Using Memcached**

Installing Memcached alone is no panacea; you have to do some work to use it. Profile your application and database queries to see where you're killing the most time and then cache from there. You also have to handle updating and purging your cache, because immediate cache coherency is important for most applications. If your application's internal API is already pretty clean, and you don't haphazardly hit the database all over your code, adding Memcached support should be easy. In your getter functions, simply try Memcached first. On a miss, hit the database and then populate Memcached. In your setter functions, update both the database and Memcached. You may find race conditions and cache coherency problems to deal with, but the
Memcached API provides means to deal with them.

Memcached also is useful for storing data you don't really need to put on disk. For example, LiveJournal uses it to prevent accidental duplicate submissions of requests by storing the transaction's result code in Memcached, keyed by a transaction signature. Another example of Memcached as a primary data store, as opposed to a cache, is warding off dumb and/or malicious bots, often spammers. By keeping track of the last times and actions of each IP address and session, our code automatically can detect patterns and notify us of attacks early on, taking automatic action as necessary. Storing this information in the database would've been wasteful, burdening the disks unnecessarily. Putting it in memory is fine, however, because the data is safe to lose if a Memcached node fails.

I asked the mailing list what interesting things they're using Memcached for, and here's what they said:

- Many people use it like we do on LiveJournal, as a typical cache for small Web objects.
- One site is using it to pass the currently playing song from their Java streaming server to their PHP Web site. They used to use a database for this, but they report hitting Memcached is much nicer.
- A lot of people are caching authentication info and session keys.
- One person reported speeding up mail servers by caching known good and known bad hosts and authentication details.

I continue to receive interesting e-mails and suggestions, so I'm happy that people are finding good uses for it.

**Alternatives**

If you can get away with running your threaded application on a single machine and have no use for a global cache, you probably don't need Memcached. Likewise, SysV shared memory may work for you, if you're sitting on a single machine.

A few people have suggested that MySQL 4.x's query cache might negate the need for Memcached. The MySQL query cache is emptied every time a relevant table is updated in any way. It's mostly a feature useful for read-only sites. LiveJournal is incredibly write-heavy, as are most high-traffic sites nowadays. Also, as with other databases, the MySQL caches together can't exceed the maximum address space the kernel provides, usually 3GB on a 32-bit machine, which gets to be cramped.

Another option for some people is MySQL's in-memory table handler. It wasn't attractive for my uses because it's limited to fixed-length records, not allowing BLOB or TEXT columns. The total amount of data you can store in it also is limited, so we still would've needed to run a bunch of them and fan out keys amongst them.

**Acknowledgements**

I'd like to thank Anatoly Vorobey for all of his hard work on the Memcached server, Lisa Phillips for putting up with early crash-prone versions and all the users on the mailing list who have sent in patches, questions and suggestions.

**Resources for this article:** [www.linuxjournal.com/article/7559](http://www.linuxjournal.com/article/7559).
Brad Fitzpatrick has been hacking database-driven Web sites for eight years. In addition to riding his bike, Brad enjoys trying to think up alternative solutions to problems that otherwise might involve salespeople. Unless you're pitching blue pills or informing him of dead servers, Brad welcomes your mail at 

brad@danga.com.